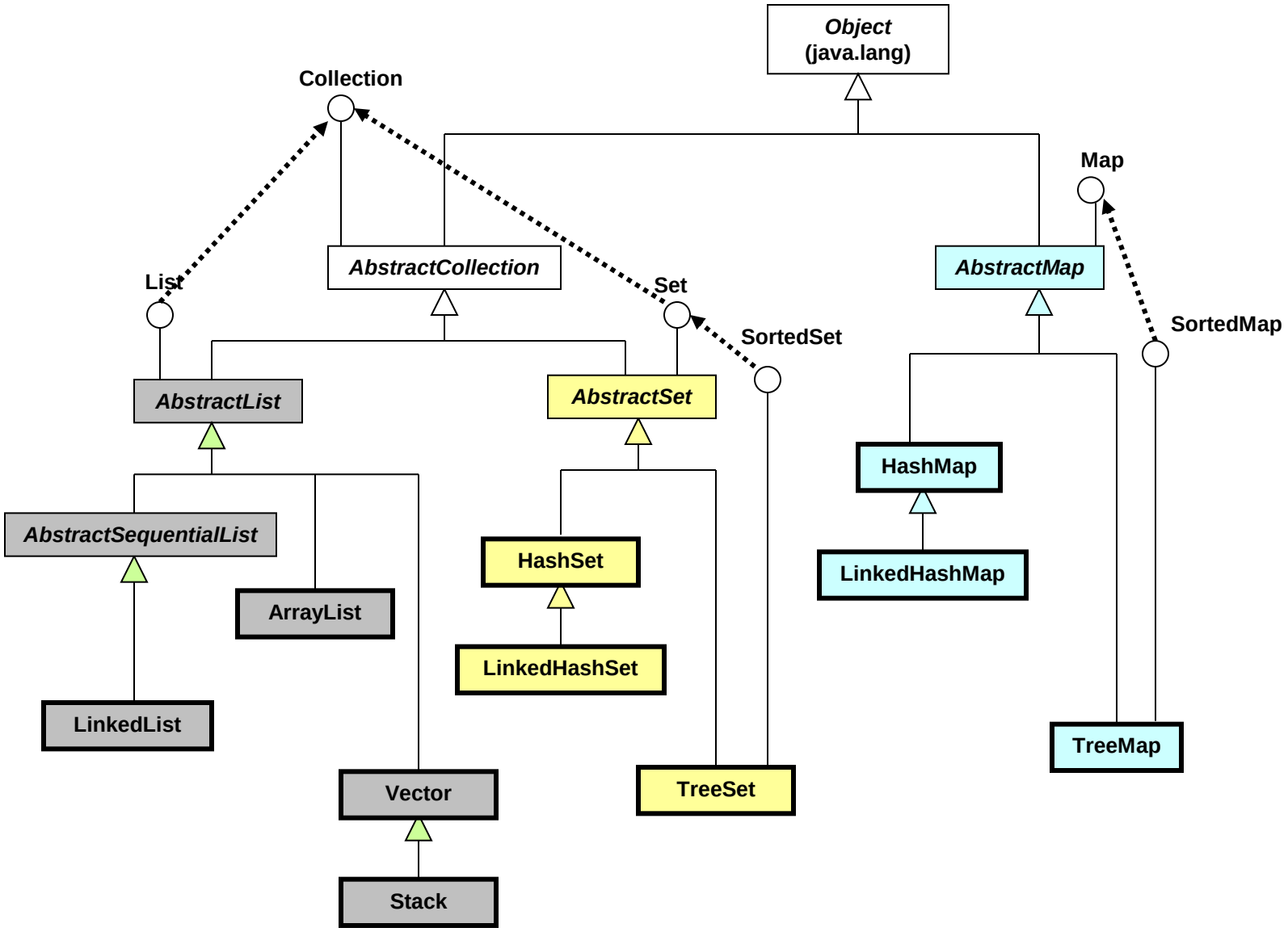
Week 11
Java Collection Framework

(This topic is not examinable)

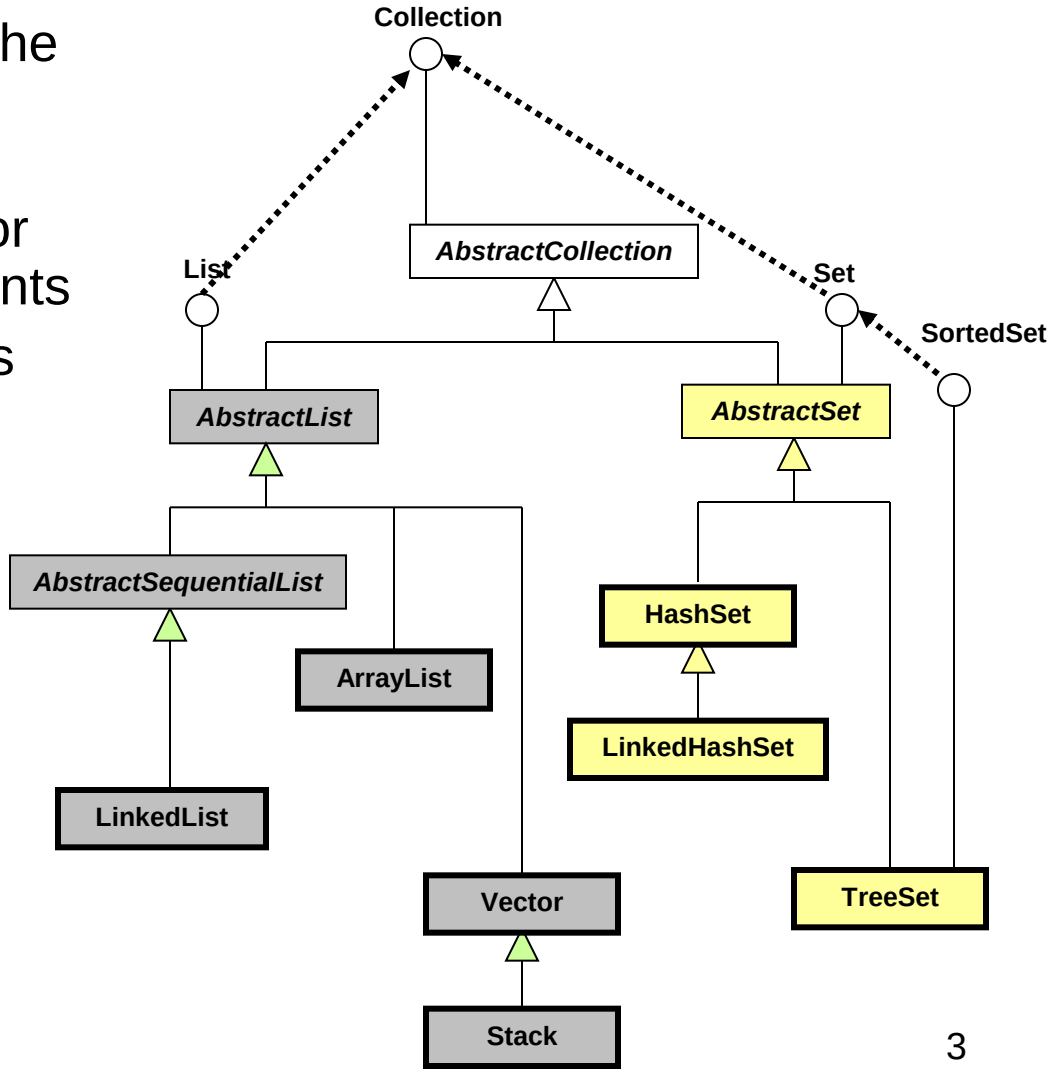
Read
Pages 714 – 726
(optional)

Relationship between Java Interfaces and Classes



Collection Interface

- The Collection interface is the root of the Collection hierarchy.
- Provides basic operations for adding and removing elements
- The AbstractCollection class provides partial implementation of the Collection interface.
- Some Collection implementations
  - allow duplicate elements
  - allow elements to be ordered
  - provide synchronized methods



Collection Interface

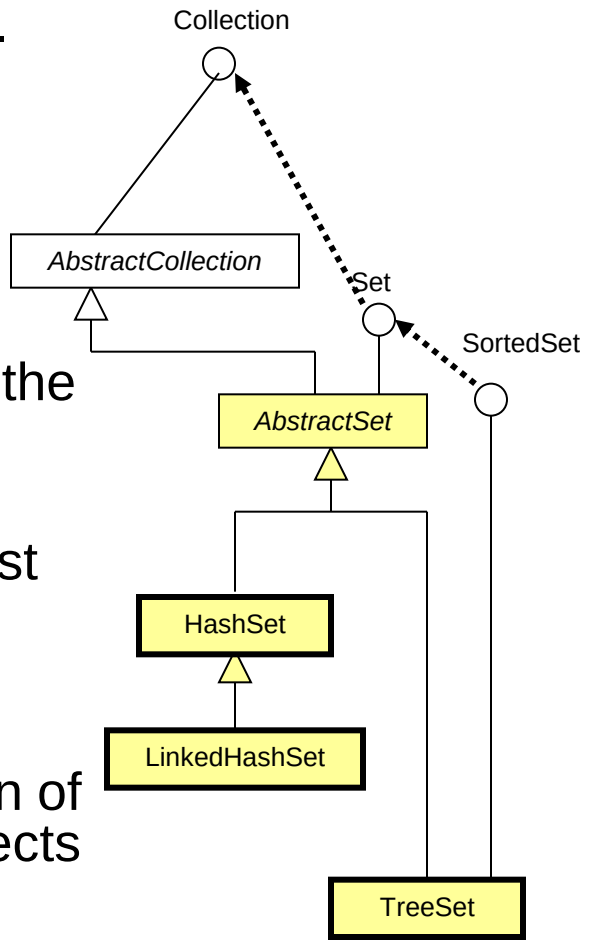
boolean add(Object o)	adds a new element
boolean addAll(Collection c)	adds all elements in c
void clear()	removes all elements in collection
boolean contains(Object o)	returns true if collection contains o
boolean containsAll(Collection c)	returns true if collection contains all elements in c
Iterator iterator()	returns an Iterator for elements in collection
boolean remove(Object o)	removes element o from collection
boolean removeAll(Collection c)	removes all element in c from collection
int size()	returns the number of elements in collection
Object[] toArray()	returns an array of Object of collection elements
Object[] toArray(Object[] array)	returns an array of Object of specified type
boolean retainsAll(Collection c)	retains elements common to collection and c

Iterator Interface

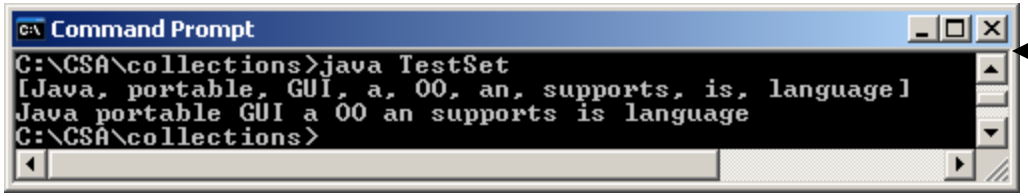
boolean hasNext()	returns true if iterator has more elements
Object next()	returns next element
void remove()	removes last element obtained by next()

Sets

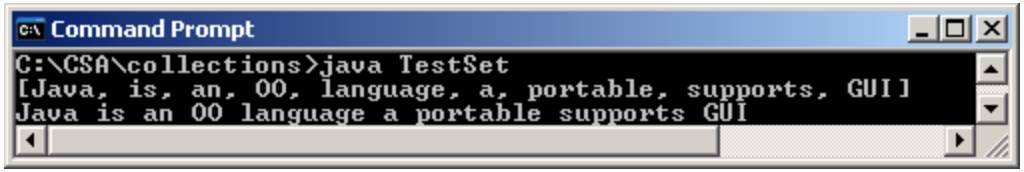
- Set extends Collection but has no new methods.
- But classes implementing Set interface must ensure no duplicates. For example, it cannot contain e1 and e2 if e1.equals(e2) is true.
- HashSet provides a concrete implementation of the Set. Elements in HashSet are not ordered.
- LinkedHashSet extends HashSet with a linked list and supports an ordering. Elements can be retrieved in order of insertion.
- TreeSet class provide a concrete implementation of SortSet. It guarantees elements are sorted. Objects inserted must be comparable with each other.
- Next program compares the output for the three concrete classes implementing the Set interface.



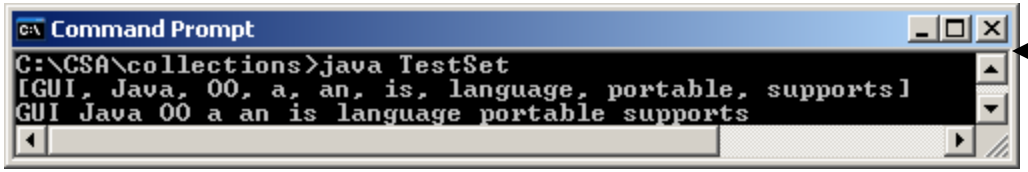
```
import java.util.*;
public class TestSet
{
   public static void main(String args[])
   {
      Set set = new HashSet();
       // Set set = new LinkedHashSet();
       // Set set = new TreeSet();

      String text = "Java is an OO language. Java is a portable language. Java supports GUI.";

      StringTokenizer st = new StringTokenizer(text," .");
      while (st.hasMoreTokens())
         set.add(st.nextToken());
      System.out.println(set);

      Iterator iterator = set.iterator();
      while ( iterator.hasNext()) {
         System.out.print(iterator.next() + " ");
      }
   }
}
```



Using HashSet



Using LinkedHashSet

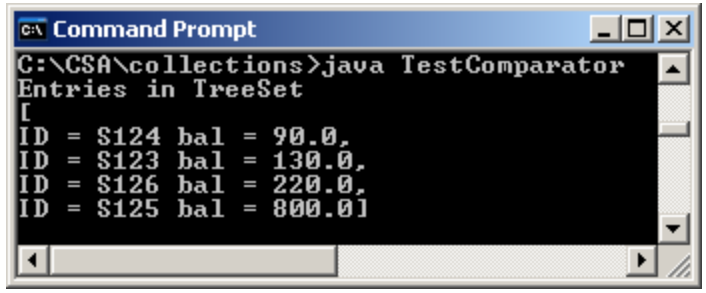


Using TreeSet

The Comparator Interface

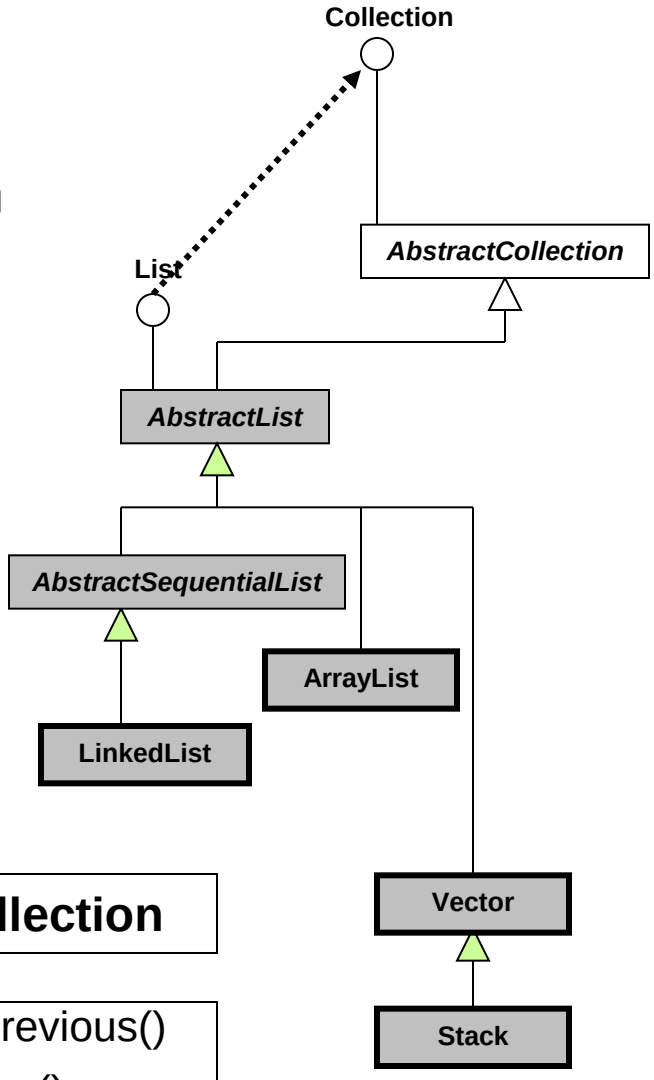
- To insert objects belonging to different classes in a TreeSet define a Comparator. You may also use this interface when classes do not implement the Comparable interface (compareTo method) or to override the normal behavior of their compareTo() methods.
- This interface has two methods:
  - public int compare(Object e1, Object e2)
  - public boolean equals(Object e)
- The equals() method can be omitted as it is also defined in Object.
- The next program stores the Account objects in a TreeSet sorted based on their account balances. The comparison is based on the AccountsComparator class.
- If the default constructor is used for creating a TreeSet object the compareTo method will be used for comparison. If the constructor TreeSet(Comparator c) is used instead, the compare method of c will be used for comparison.

```
import java.util.*;
class Account
{
   private String ID;
   private double balance;
   public Account(String ID, double balance)
   {  this.ID = ID;
      this.balance = balance;
   }
   public String getID() { return ID; }
   public double balance() { return balance; }
   public String toString()
   {  return new String("ID = "+ID+ " bal = " + balance + \n");
   }
}

class AccountsComparator implements Comparator
{  public int compare(Object o1, Object o2)
   {
      double b1 = ((Account) o1).balance();
      double b2 = ((Account) o2).balance();
      int val = b1>b2 ? 1: b1==b2 ? 0 : -1;
      return val;
   }
}

public class TestComparator
{
   public static void main(String[] args)
   {
      TreeSet tSet = new TreeSet(new AccountsComparator());
      tSet.add(new Account("S123",130.0));
      tSet.add(new Account("S124",90.0));
      tSet.add(new Account("S126",220.0));
      tSet.add(new Account("S125",800.0));

      System.out.println("Entries in TreeSet");
      System.out.println(tSet);
   }
}
```



Lists

- Unlike a set a list allow duplicate elements
- List allows users to access elements based on their position or index. It also allows users to store elements in specific index.
- The list interface adds index related operations such as add(int index, Object o).
- It specifies an iterator that allows bidirectional traversing.



java.util.Collection

java.util.List

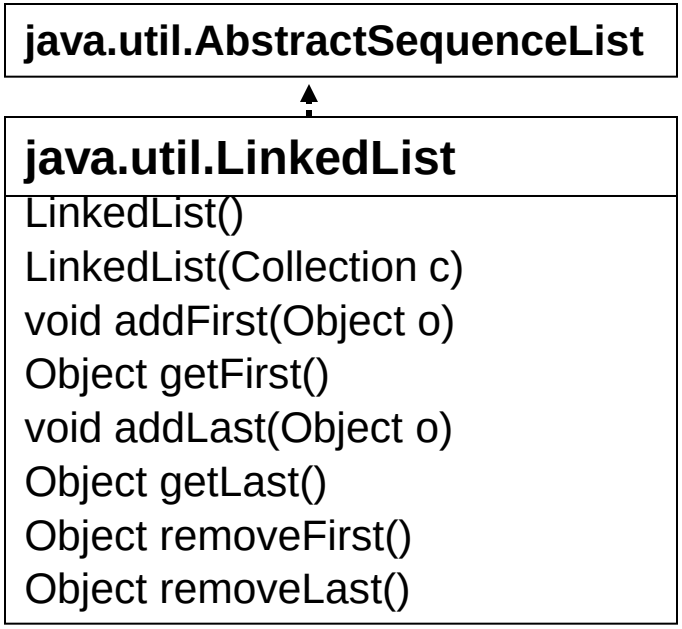
boolean add(int index, Object e)
Object get(int index)
Int indexOf(Object element)
ListIterator listIterator(
ListIterator listIterator(int startIndex)
List subList(int from, int to)
Object remove(int index)

java.util.Collection

boolean hasPrevious()
Object previous()
int previousIndex()
int nextIndex()

LinkedList class

- Stores objects in a linked list
- This class is more suited than ArrayList when elements are added and deleted anywhere in the list as it can grow and shrink dynamically.
- This class provides additional methods for retrieving, inserting and deleting elements.

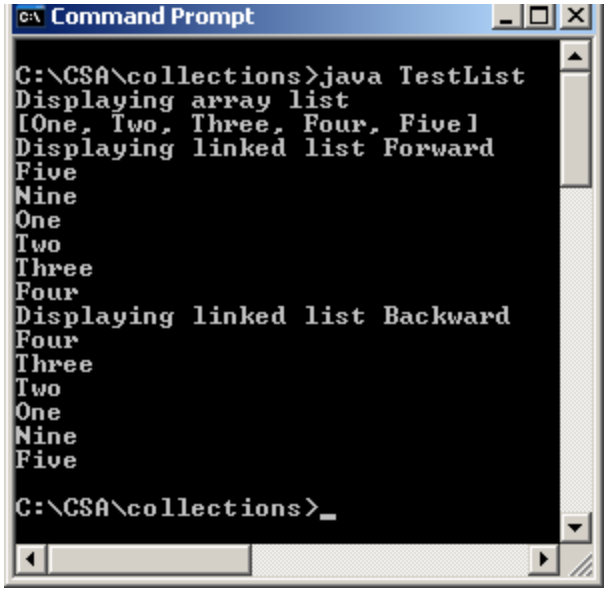


ArrayList class

- It stores elements in an array which grows dynamically. Each ArrayList instance has a capacity which stores the current size of the array. As more elements are added and the number of elements exceed the capacity new arrays are created and the elements copied.Hence in addition to implementing the List interface ArrayList class provides methods to manipulate the size of the array.
- ArrayList provides the most efficient data structure If random access of elements are required but with little or no insertions and deletions, except at the end.
- An ArrayList instance can be created with
  - no arguments
  - using an existing collection
  - passing the initial capacity.
- The next sample program creates an ArrayList object inserts and removes a few String objects. The program also creates an instance of a linked list and inserts and removes some elements before traversing the list backwards.

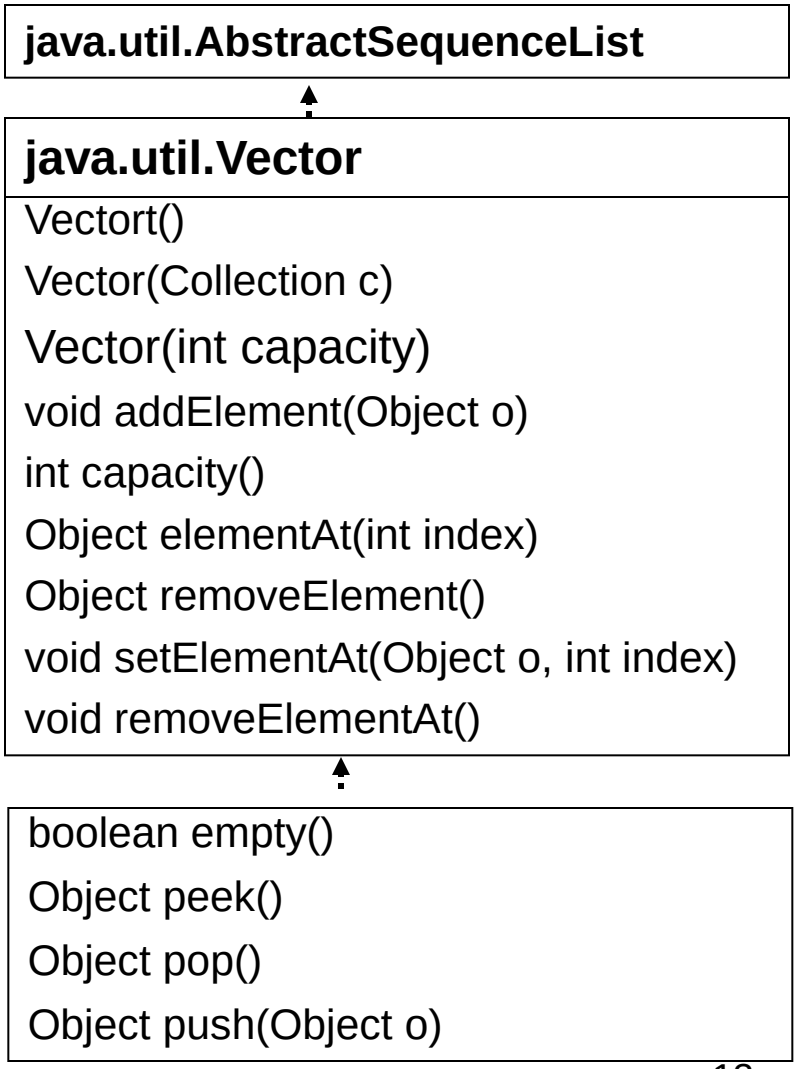
Sample program using ArrayList & LinkedList

```
import java.util.*;
public class TestList
{
   public static void main(String args[])
   {
      List arrayList = new ArrayList();
      arrayList.add(new String("One"));
      arrayList.add(new String("Three"));
      arrayList.add(new String("Five"));
      arrayList.add(new String("Four"));

      arrayList.add(1,new String("Two"));
      Object o = arrayList.remove(3);
      arrayList.add(4,o);
      System.out.println("Displaying array list");
      System.out.println(arrayList);

      LinkedList lList = new LinkedList(arrayList);
      lList.addFirst(lList.removeLast());
      lList.add(1,new String("Nine"));

      System.out.println("Displaying linked list Forward");
      ListIterator lIt = lList.listIterator();
      while (lIt.hasNext())
         System.out.println(lIt.next());

      System.out.println("Displaying linked list Backward");
      lIt = lList.listIterator(lList.size());
      while (lIt.hasPrevious())
         System.out.println(lIt.previous());
   }
}
```



Vector and Stack classes

- Vector and Stack classes were introduced before JCF was introduced. Though they are redesigned to fit JCF hierarchy the original methods are retained
- The Vector class is the same as ArrayList, except that it contains thread safe methods for manipulating a Vector object.
- The Stack class is a subclass of Vector. The pop() removes the top element. The push(
- Object o) adds the element o to the stack. The peek() returns the top element without removing it.



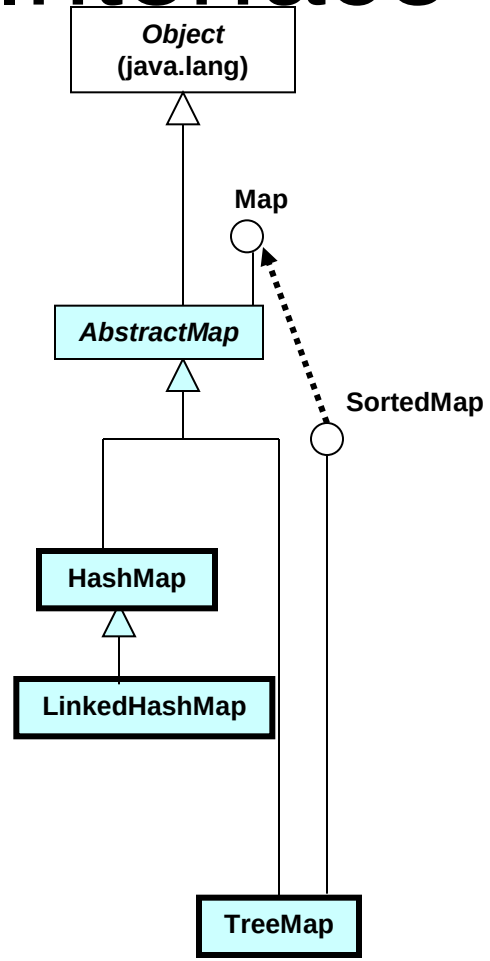
Maps

- Unlike collection interface (sub interfaces Set and List), Map interface maps a key to a value.
- The key can be any object but duplicate keys are not allowed.
- Each key can map to only one value
- The Map interface provides the methods for updating and querying.

void clear()	removes all mappings
boolean containsKey(Object key)	returns true if there is a mapping for this key
boolean containsValue(Object value)	returns true if there key mapping to this value
Set entrySet()	a set consisting of the entries { k1=v1, k2=v2, …}
Object get(Object key)	returns the value for specified key
boolean isEmpty()	returns true if map is empty false otherwise
Set keySet()	returns a set consisting of all the keys
Object put(Object key, Object value)	puts the mapping (key, value) in the map
void putAll(Map m)	puts all the mapping from m
Object remove(Object key)	removes the mapping for specified key
int size()	returns the number of mappings
Collection values()	returns a collection consisting of all the values

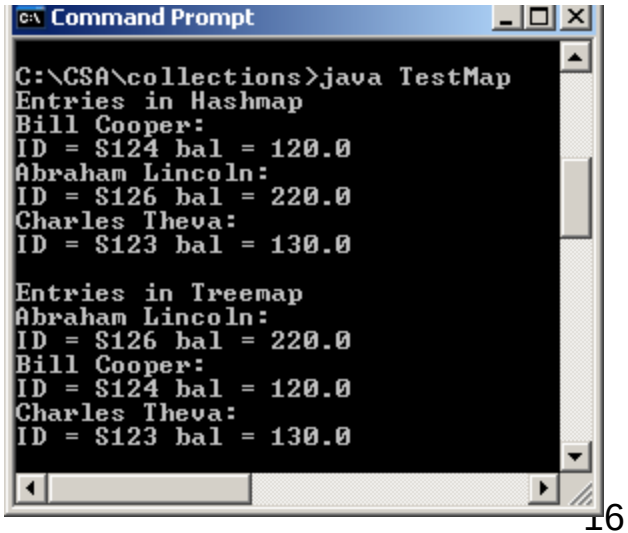
Implementation of Map Interface

- HaspMap
  - Efficient for locating a value, inserting/deleting mapping
  - Entries not ordered
  - Pre JDK 1.2 equivalent is HashTable (redesigned to fit JCF but methods retained for compatibility)
- Linked HashMap
  - Extends HashMap with a linked list
  - Can be accessed in order of insertion/access
- TreeMap
  - Implements the interface SortedMap and efficient for accessing maps in sorted order
  - Comparison can be done using the compareTo() method (default) or by passing a Comparator to the TreeMap constructor



Sample program using HaspMap & TreeMap

```
import java.util.*;
class Account
{  private String ID;
   private double balance;
   public Account(String ID, double balance)
   {  this.ID = ID;
      this.balance = balance;
   }
   public String getID() { return ID; }
   public double balance() { return balance; }
   public String toString()
   {  return new String("ID = "+ID+ " bal = " + balance);
   }
}
public class TestMap
{  public static void main(String[] args)
   {  HashMap hashMap = new HashMap();
      hashMap.put("Charles Theva", new Account("S123",130.0));
      hashMap.put("Bill Cooper", new Account("S124",90.0));
      hashMap.put("Abraham Lincoln", new Account("S126",220.0));
      hashMap.put("Bill Cooper", new Account("S124",120.0));
      System.out.println("Entries in Hashmap");
      displayMap(hashMap);

      System.out.println("\nEntries in Treemap");
      TreeMap treeMap = new TreeMap(hashMap);
      displayMap(treeMap);
   }
   public static void displayMap(Map m)
   {
      Set keySet = m.keySet();
      Iterator iterator = keySet.iterator();
      while (iterator.hasNext()) {
         String key = (String) iterator.next();
         System.out.println(key + ":" + m.get(key) );
      }
   }
}
```

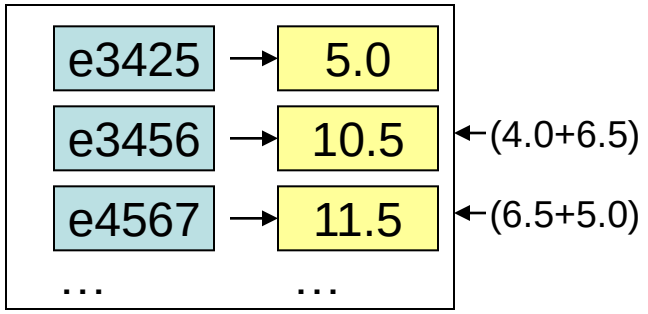


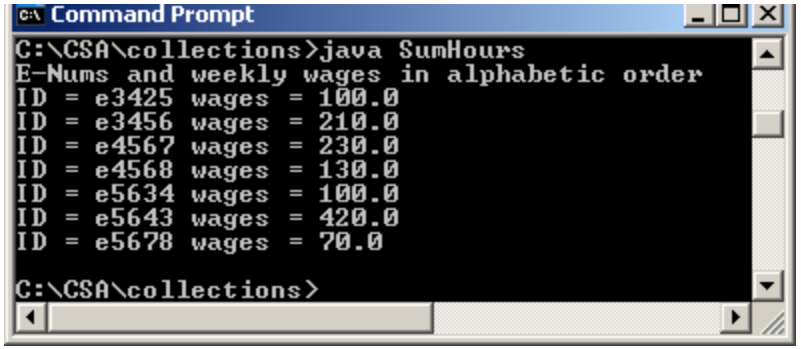
A simple program using a hash map

- The program reads the daily hours worked by the casual employees from the file casual.txt.
- It then sums up the hours for each employee storing them in a hasp map.
- It then computes the weekly wages by computing the total hours for each employee by hourly rate.

casual.txt

e4567	6.5
e5643	8.0
e3456	6.5
e5634	5.0
e5678	3.5
e4568	6.5
e5643	8.0
e3425	5.0
e4567	5.0
e3456	4.0
e5643	5.0





A simple program using a hash map

```
import java.io.*;
import java.util.*;
public class SumHours
{
   public static void main(String args[]) throws IOException
   {  // Create a hash map to hold emplyee ID and total hours
      Map hashMap = new HashMap();
      BufferedReader b = new BufferedReader(new FileReader("casual.txt"));
      String line;
      while ((line = b.readLine()) != null)
      {
         StringTokenizer st = new StringTokenizer(line);
         while ( st.countTokens() >= 2)
         {
            String ID = st.nextToken();
            double hour = Double.parseDouble(st.nextToken());
            if (hashMap.get(ID) != null)
            {
               double totalHours = ((Double) hashMap.get(ID)).doubleValue();
               totalHours += hour;
               hashMap.put(ID,new Double(totalHours));
            }
            else hashMap.put(ID,new Double(hour));
         }
      }
      b.close();
```

A simple program using a hash map

```
 // Create a tree map paasing the hash map created
      Map treeMap = new TreeMap(hashMap);

         // Get an entry set for the tree map
      Set keys = treeMap.keySet();
      // Get an iterator for the keys
      Iterator iterator = keys.iterator();

      System.out.println("E-Nums and weekly wages in alphabetic order");
      while (iterator.hasNext())
      {
         String ID = (String) iterator.next();
         double hours = ((Double) treeMap.get(ID)).doubleValue();
         double wages =  hours * 20.0;
         System.out.println("ID = " +ID+" wages = " + wages );
      }
   }
}
```

The Collections class

- Provides static methods for operating on collections and maps.
- Allows creation of thread safe and read-only collection and map classes
  - List synchronizedList(List list), Map synchronizedMap(Map m)
  - List unmonifiableList(List list), Map unmonifiableMap(Map m)
- Allows sorting using Comparable or Comparator interfaces
  - void sort(List list) void sort(List list, Comparator c)
- Allows binary search of sorted lists
- Provides methods for
  - copying lists
  - filling lists
  - finding minimum and maximum in collections

The Arrays class

- The Arrays class provides various static methods for arrays including primitive arrays
  - Sorting (overloaded for int, long, double …)
  - Filling (overloaded for int, long, double …)
  - Searching (overloaded for int, long, double …)

Sample program using Arrays class

```
import java.util.*;
public class TestClassArrays
{
   public static void main(String[] args)
   {
      int[] nums = { 1,2,3,4,5,6,7,8};
      Arrays.fill(nums,1,4,8);  // sets elements at index 1,2 and 3 to 8
      print("After fill",nums);
      Arrays.sort(nums);
      print("After sorting",nums);
      int index = Arrays.binarySearch(nums,3);
      System.out.println("BSearch found 3 at "+index);
      index = Arrays.binarySearch(nums,9);
      System.out.println("BSearch found 9 at "+index);
   }
   public static void print(String message, int[] a)
   {
      System.out.println(message);
      for (int i=0; i<a.length; i++)
         System.out.print(a[i] + " ");
      System.out.println();
   }
}
```